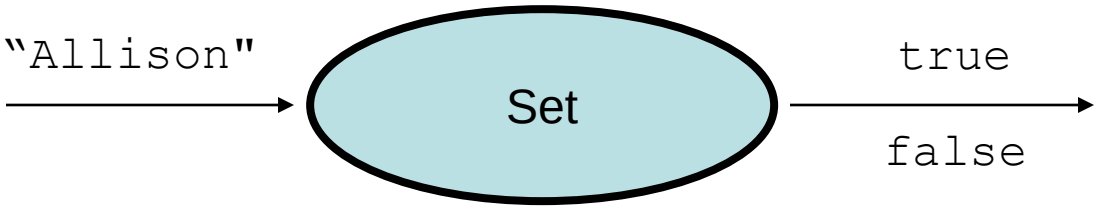
CSE 143

Lecture 12: Sets and Maps

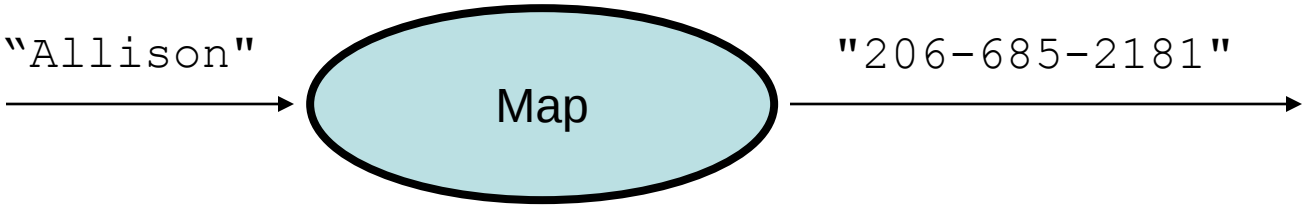
reading: 11.2 - 11.3

Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - *Set: Is Allison found in the set? (true/false)*



  - *Map: What is Allison's phone number?*



Problem: opposite mapping

- It is legal to have a map of sets, a list of lists, etc.
- Suppose we want to keep track of each TA's GPA by name.

```
Map<String, Double> taGpa = new HashMap<String, Double>();
taGpa.put("Melissa", 3.6);
taGpa.put("Ying", 4.0);
taGpa.put("Vivyan", 2.9);
taGpa.put("Rajas", 3.6);
taGpa.put("Jenny", 2.9);
...
System.out.println("Rajas's GPA is " +
                   taGpa.get("Rajas"));   // 3.6
```

- This doesn't let us easily ask which TAs got a given GPA.
  - How would we structure a map for that?

Reversing a map

- We can reverse the mapping to be from GPAs to names.

```
Map<Double, String> taGpa = new HashMap<Double, String>();
taGpa.put(3.6, "Melissa");
taGpa.put(4.0, "Ying");
taGpa.put(2.9, "Vivyan");
taGpa.put(3.6, "Rajas");
taGpa.put(2.9, "Jenny");
...
System.out.println("Who got a 3.6? " +
                   taGpa.get(3.6));   // ???
```

- What's wrong with this solution?
  - More than one TA can have the same GPA.
  - The map will store only the last mapping we add.

Proper map reversal

- Really each GPA maps to a *collection* of people.

```
Map<Double, Set<String>> taGpa =
        new HashMap<Double, Set<String>>();
taGpa.put(3.6, new TreeSet<String>());
taGpa.get(3.6).add("Melissa");
taGpa.put(4.0, new TreeSet<String>());
taGpa.get(4.0).add("Ying");
taGpa.put(2.9, new TreeSet<String>());
taGpa.get(2.9).add("Vivyan");
taGpa.get(3.6).add("Rajas");
taGpa.get(2.9).add("Jenny");
...
System.out.println("Who got a 3.6? " +
                   taGpa.get(3.6));  //[Melissa, Rajas]
```

  - must be careful to initialize the set for a given GPA before adding

Exercise

- Modify the word count program to print every word that appeared in the book at least 1000 times, in sorted order from least to most occurrences.

Examining sets and maps

- elements of Java `Sets` and `Maps` can't be accessed by index
  - must use a "foreach" loop:

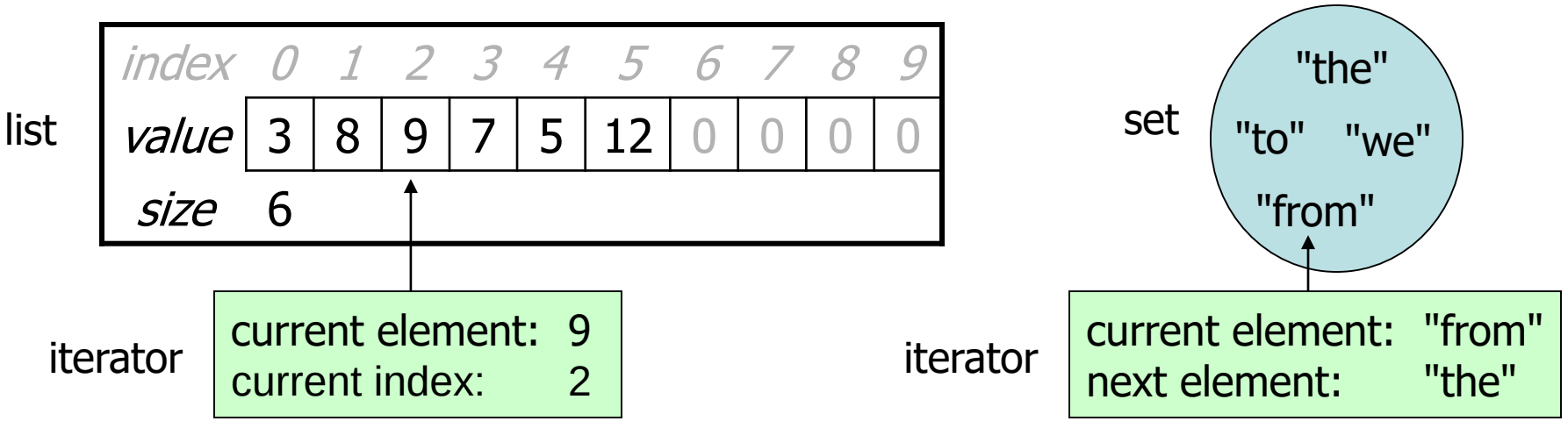
```
Set<Integer> scores = new HashSet<Integer>();
for (int score : scores) {
    System.out.println("The score is " + score);
}
```

  - Problem: foreach is read-only; cannot modify set while looping

```
for (int score : scores) {
    if (score < 60) {
    // throws a ConcurrentModificationException
        scores.remove(score);
    }
}
```

Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position



`Iterator` methods

`hasNext()`	returns `true` if there are more elements to examine
`next()`	returns the next element from the collection (throws a `NoSuchElementException` if there are none left to examine)
`remove()`	removes the last value returned by `next()` (throws an `IllegalStateException` if you haven't called `next()` yet)

- `Iterator` interface in `java.util`
  - every collection has an `iterator()` method that returns an iterator over its elements

```
Set<String> set = new HashSet<String>();
…
Iterator<String> itr = set.iterator();
…
```

Iterator example

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);    // Will
scores.add(87);
scores.add(43);    // Allison
scores.add(72);
…

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();

    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores);  // [72, 87, 94]
```

Iterator example 2

```
Map<String, Integer> scores = new TreeMap<String, Integer>();
scores.put("Will", 38);
scores.put("Natalie", 94);
scores.put("Chloe", 87);
scores.put("Allison", 43);
scores.put("Sarang", 72);
…

Iterator<String> itr = scores.keySet().iterator();
while (itr.hasNext()) {
    String name = itr.next();
    int score = scores.get(name);
    System.out.println(name + " got " + score);

    // eliminate any failing students
    if (score < 60) {
        itr.remove();      // removes name and score
    }
}
System.out.println(scores); //{Chloe=87, Natalie=94, Sarang=72}
```

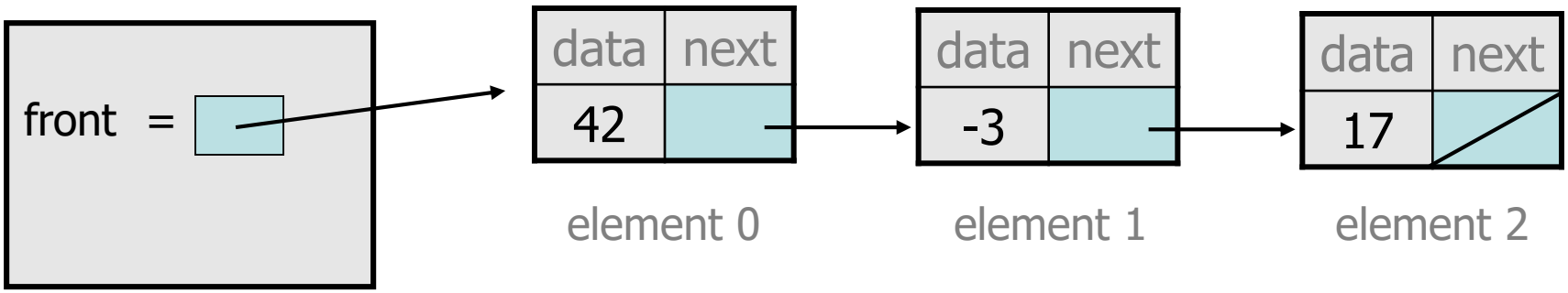
A surprising example

- What's bad about this code?

```
List<Integer> list = new LinkedList<Integer>();
```

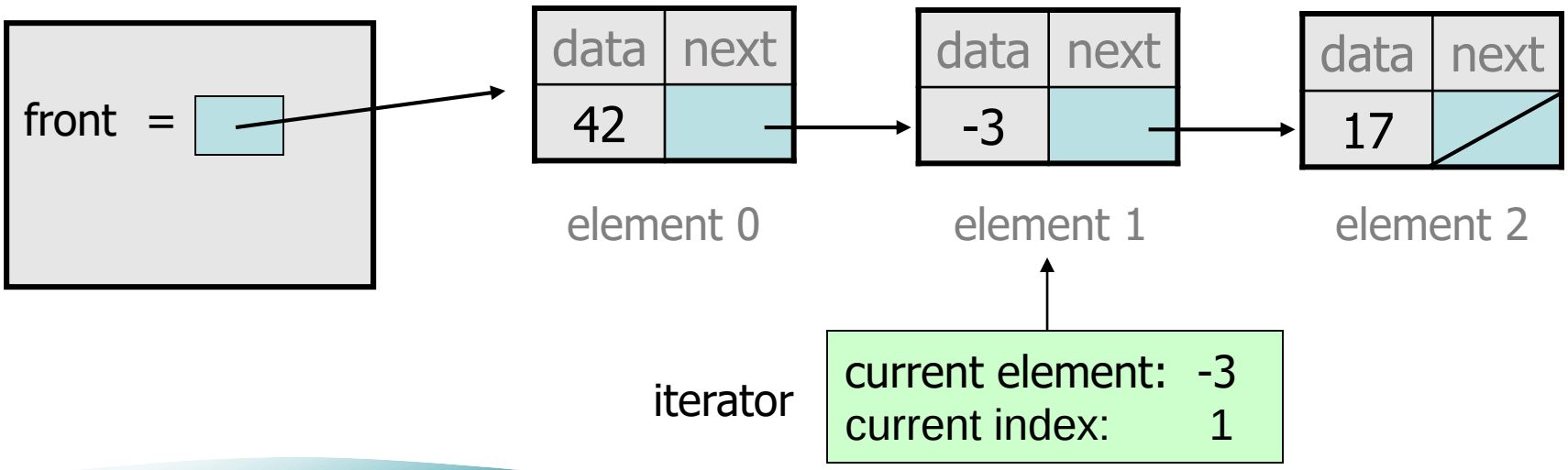
... (add lots of elements) ...

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```



Iterators and linked lists

- Iterators are particularly useful with linked lists.
  - The previous code is O(N^2) because each call on `get` must start from the beginning of the list and walk to index `i`.
  - Using an iterator, the same code is O(N). The iterator remembers its position and doesn't start over each time.



Exercise

- Modify the Book Search program from last lecture to eliminate any words that are plural or all-uppercase from the collection.

- Modify the TA quarters experience program so that it eliminates any TAs with 3 quarters or fewer of experience.

ListIterator

`add(`**value**`)`	inserts an element just after the iterator's position
`hasPrevious()`	`true` if there are more elements *before* the iterator
`nextIndex()`	the index of the element that would be returned the next time `next` is called on the iterator
`previousIndex()`	the index of the element that would be returned the next time `previous` is called on the iterator
`previous()`	returns the element before the iterator (throws a `NoSuchElementException` if there are none)
`set(`**value**`)`	replaces the element last returned by `next` or `previous` with the given value

```
ListIterator<String> li = myList.listIterator();
```

- lists have a more powerful `ListIterator` with more methods
  - can iterate forwards or backwards
  - can add/set element values (efficient for linked lists)